Опыт программирования задачи SSSD (Single Source Shortest Distance) в потоковой модели вычислений в парадигме раздачи

Д.Н.Змеев, А.В.Климов, Н.Н.Левченко, А.С.Окунев

Институт проблем проектирования в микроэлектронике (ИППМ) РАН, г. Москва

GraphHPC
5 марта 2014 г.
Москва, МГУ

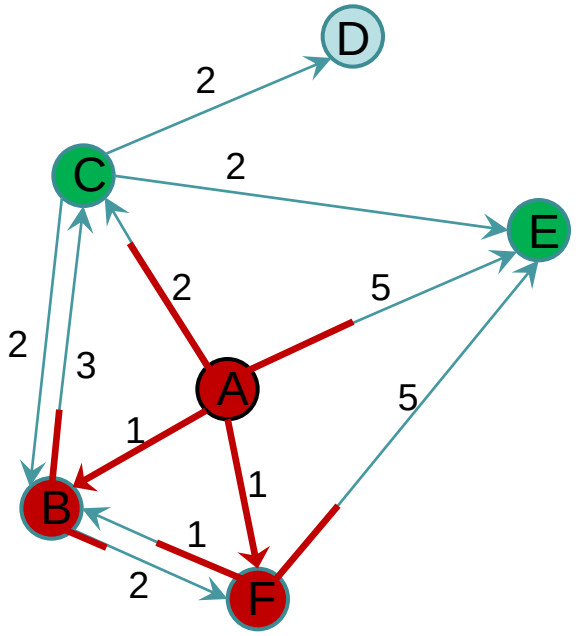
Задача SSSP (Single Source Shortest Path)

Дан граф G как множестве вершин Vert={i} и множество дуг с весами Edge(i)={(w,j)}, w>0. Многократно задается начальная вершина i_0. Требуется найти кратчайшие расстояния от i_0 до всех остальных вершин.

Принцип алгоритма Дейкстры: имитация распространения фронта волны, движущегося с единичной скоростью по дугам графа.
Физический процесс непрерывен, но алгоритм отрабатывает значимые изменения в моменты прохождения фронтом одной из вершин.

Список L ближайших проходов через вершины

(2,C)
(5,E)
(3,F)
(6,E)



$t = 3.5$

Достигнутая оценка расстояния D

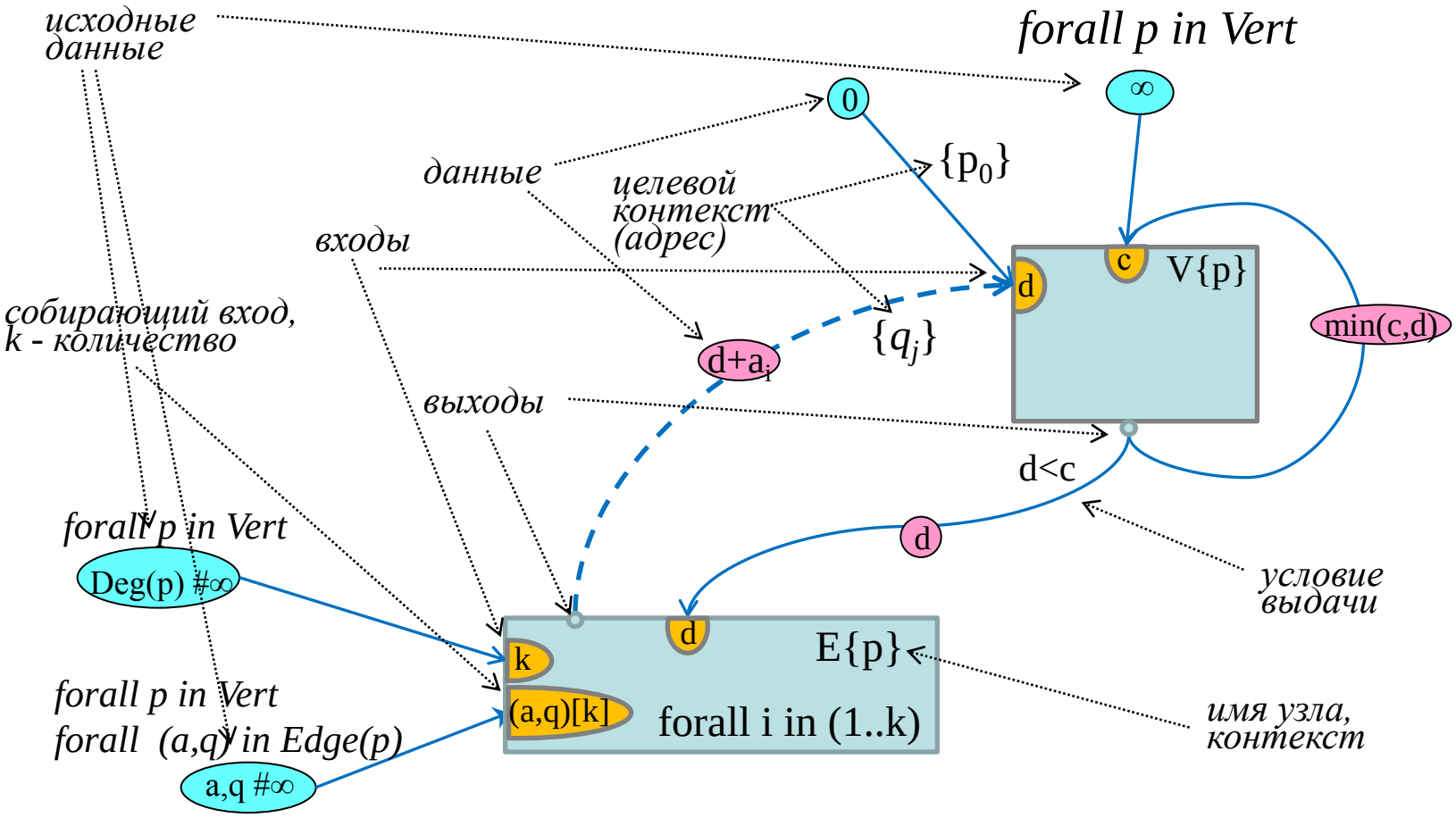
A 0
B 1
C 2
D ∞
E 5
F 1

Алгоритм

```
1.   D(:) = ∞;  -- D(i) – достигнутое расстояние до вершины i
2.   D(i0) = 0;   -- начальный элемент
3.   L:= { (0,i0) } ; -- одноэлементный список (множество)
4.   while (a,i) = popmin(L) do – выбрав элемент (a,i) с наименьшим a,
5.     if a≤D(i) then  -- доп. Relax-фильтр, здесь всегда a≥D(i)
6.       forall (w,j) in Edge(i) do
7.         b ≔ a+w; -- b – новая оценка расстояния
8.         if b<D(j) then  -- основной Relax-фильтр
9.           D(j) ≔b;
10.          append(L,(b,j));  -- вставить элемент (b,j) в список L
11.        endif
12.      endfor
13.    endif
14.  endwhile
```

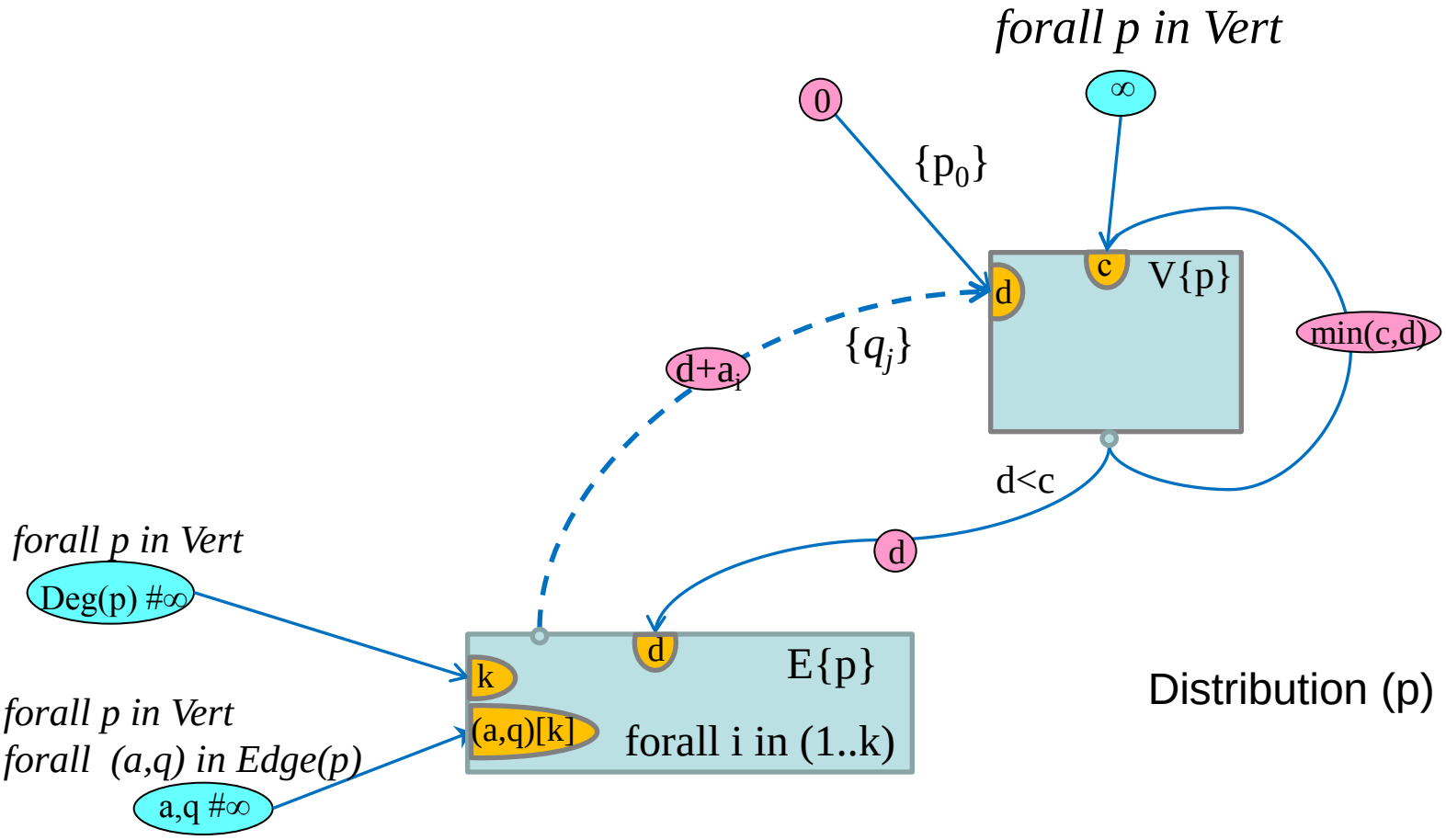
Алгоритм принципиально последовательный, тогда как физический процесс параллельный Вместе с минимальным элементом (a,i) возможно отрабатывать еще несколько следующих элементов (b,j), покуда $b<a+w_{min}$. Можно и больше, но тогда есть риск, что придется проходить по дугам повторно. И чем больше окно, тем больше параллелизма, но и больше работы. Для улучшения распараллеливаемости полезно пополнить граф так, чтобы wmin стало как можно больше (покуда позволяет объем памяти для хранения графа). Задача такого пополнения – тоже интересная задача для параллельной реализации.

Схема алгоритма. Наивный вариант (без синхронизации)



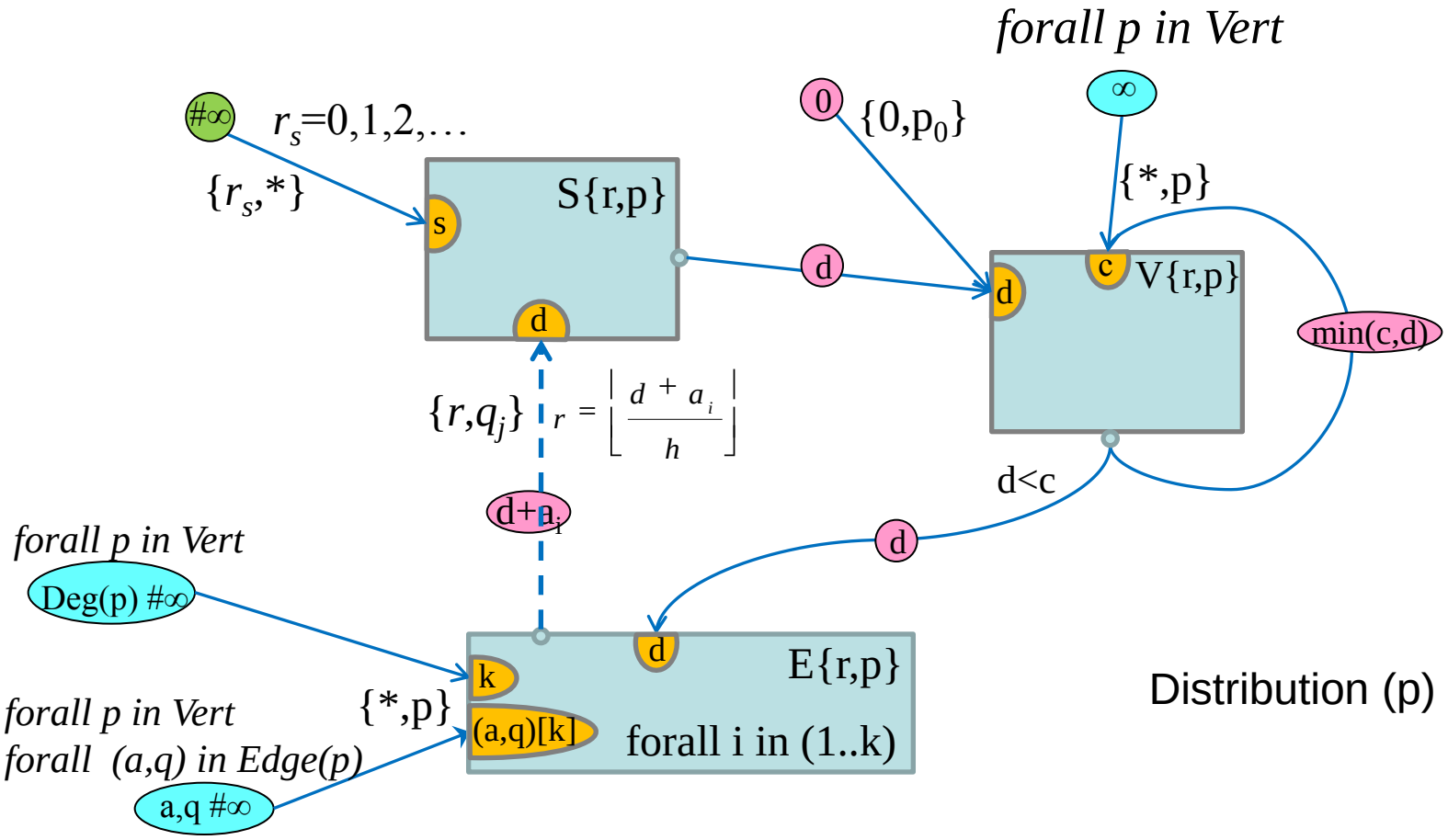
По завершению всякой активности кратчайшие расстояния от p_0 до p , будут находиться в V.c{p} . Возможны многократные проходы по дугам.
Пунктиром обозначены передачи по дугам. Только они являются нелокальными.

Схема алгоритма. Наивный вариант (без синхронизации)



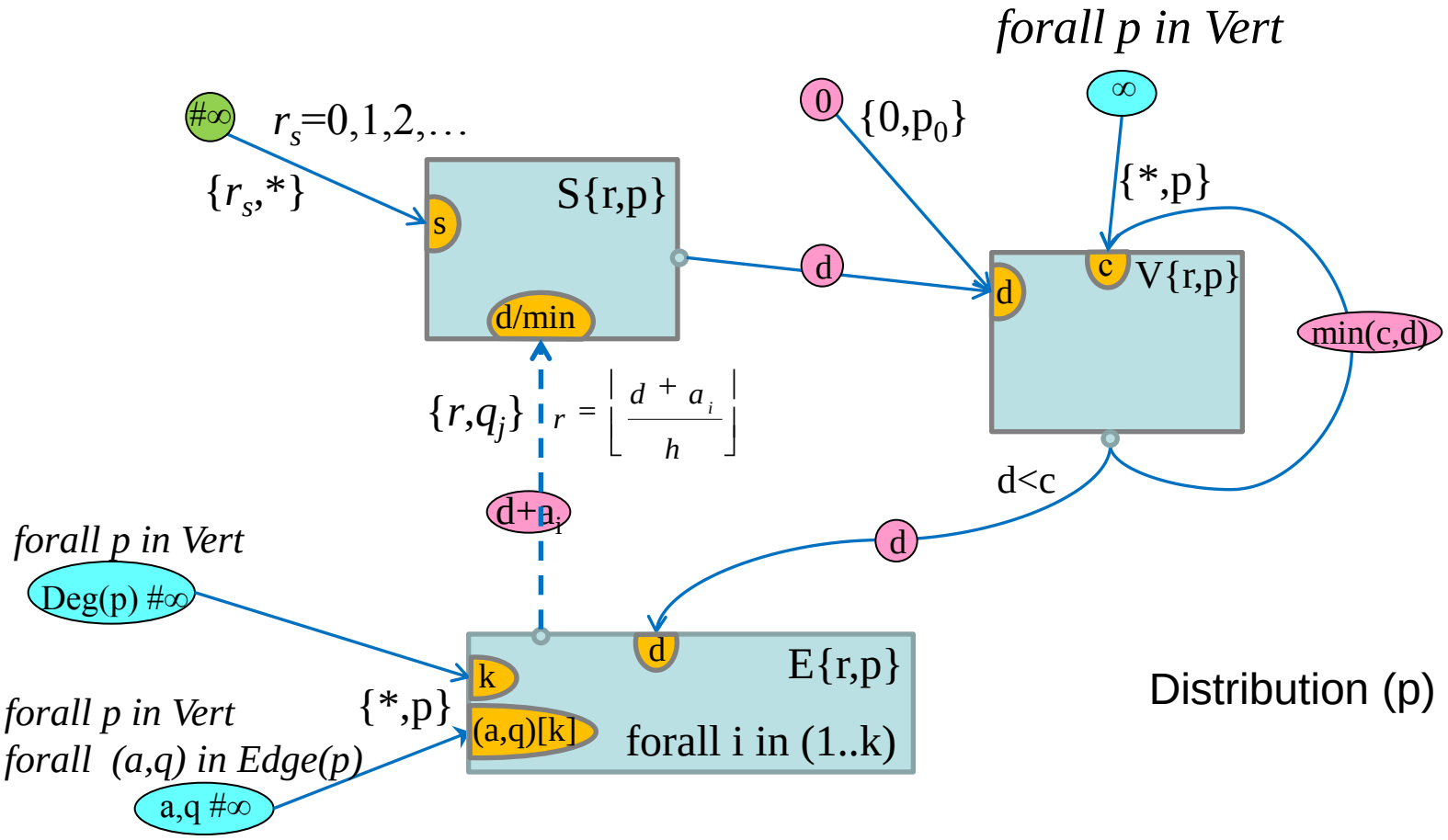
По завершению всякой активности кратчайшие расстояния от p_0 до p, будут находиться в V.c{p} . Возможны многократные проходы по дугам.
Пунктиром обозначены передачи по дугам. Только они могут быть нелокальными.
Узлы распределяются по ядрам по полю p взятием остатка от деления на число ядер N.

Схема алгоритма. Базовый вариант с синхронизацией.



Разрешающий токен S.s{*r*,*} можно посылать по завершению всякой активности в подпространстве {*r*-1,*}. Или {*r*-*W*,*}, где *W* =1,2,… – ширина окна активности.
Для этого исполняющая система должна иметь *многоуровневый детектор тишины.*
NB: здесь для токенов на узле S не требуется поиск по значению p.

Схема алгоритма. Базовый вариант с синхронизацией.



Разрешающий токен S.s{*r*,*} можно посылать по завершению всякой активности в подпространстве {*r*-1,*}. Или {*r*-*W*,*}, где *W* =1,2,… – ширина окна активности.
Для этого исполняющая система должна иметь *многоуровневый детектор тишины.*
NB: здесь для токенов на узле S не требуется поиск по значению p.
Вход S.d теперь с редукцией через *min.*

МРТ: многоуровневый распознаватель тишины (MQD: multilevel quiescence detector)

Глобальная синхронизация осуществляется посредством механизма распознавания тишины (quiescence detection), который встроен в исполняющую систему. Метод реализации – распределенные счетчики.

Простой распознаватель тишины основан на локальном подсчете токенов, посланных и принятых во всех ядрах, и глобальном суммировании этих счетчиков. Нулевая сумма является признаком тишины, то есть завершения всех активных действий.

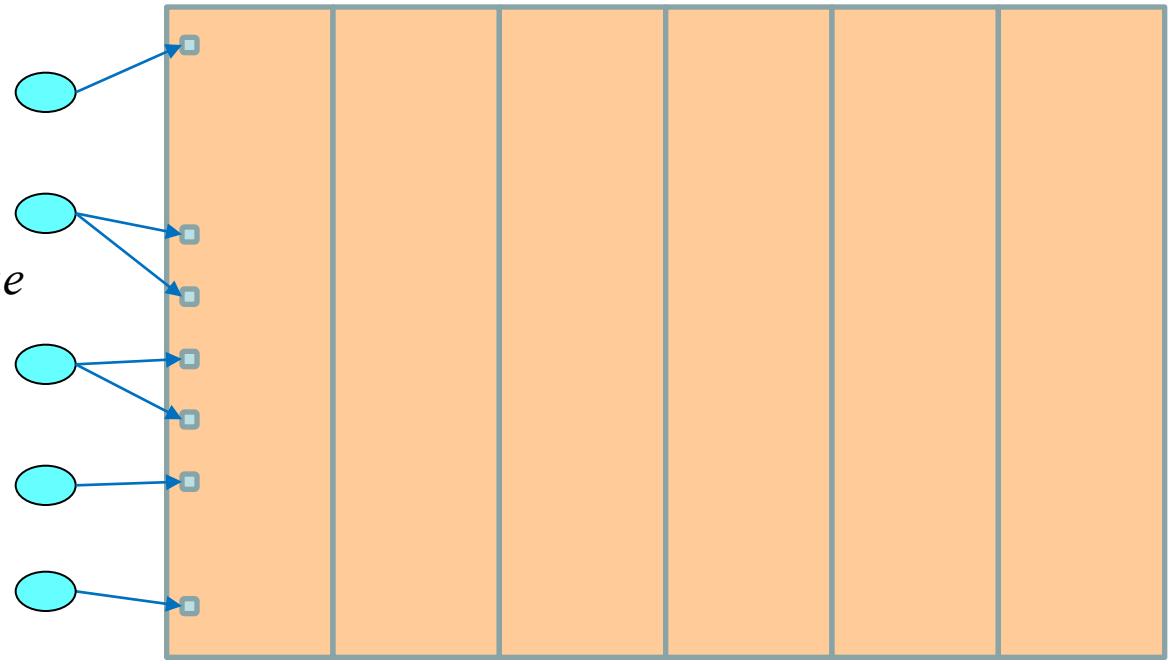
Наше решение задачи SSSD опирается на многоуровневый детектор тишины. В нем локальный подсчет ведется раздельно по всем значениям выделенного поля r. (Считается, что активность для r_1 может породить только активность для $r_2 \geq r_1$). Токены, направленные на некоторые входы (у нас это S.d), при их сохранении учитываются как посланные, но не принятые, то есть как зародыши активности. И только по их использованию (со стиранием их из памяти) они будут учтены как принятые.

Токены с r=* учитываются отдельно: их сумма виртуально прибавляется ко всем счетчикам, так что пока она не нуль, ни одна сумма для конкретного r не может быть расценена как нулевая.

МРТ глобально определяет минимальное значение r_m, в котором имеются зародыши активности, и активизирует их заданным образом (у нас это посылка #∞→S.s{r_m,*}), или сообщает, что ненулевых r больше нет. Возможна активизация сразу для W последовательных значений r, начиная с r_m: r_m, r_m+1,…, r_m+W-1 (избегая повторных и бесполезных активаций)

Ход вычисления под управлением МРТ

Активирующие токены

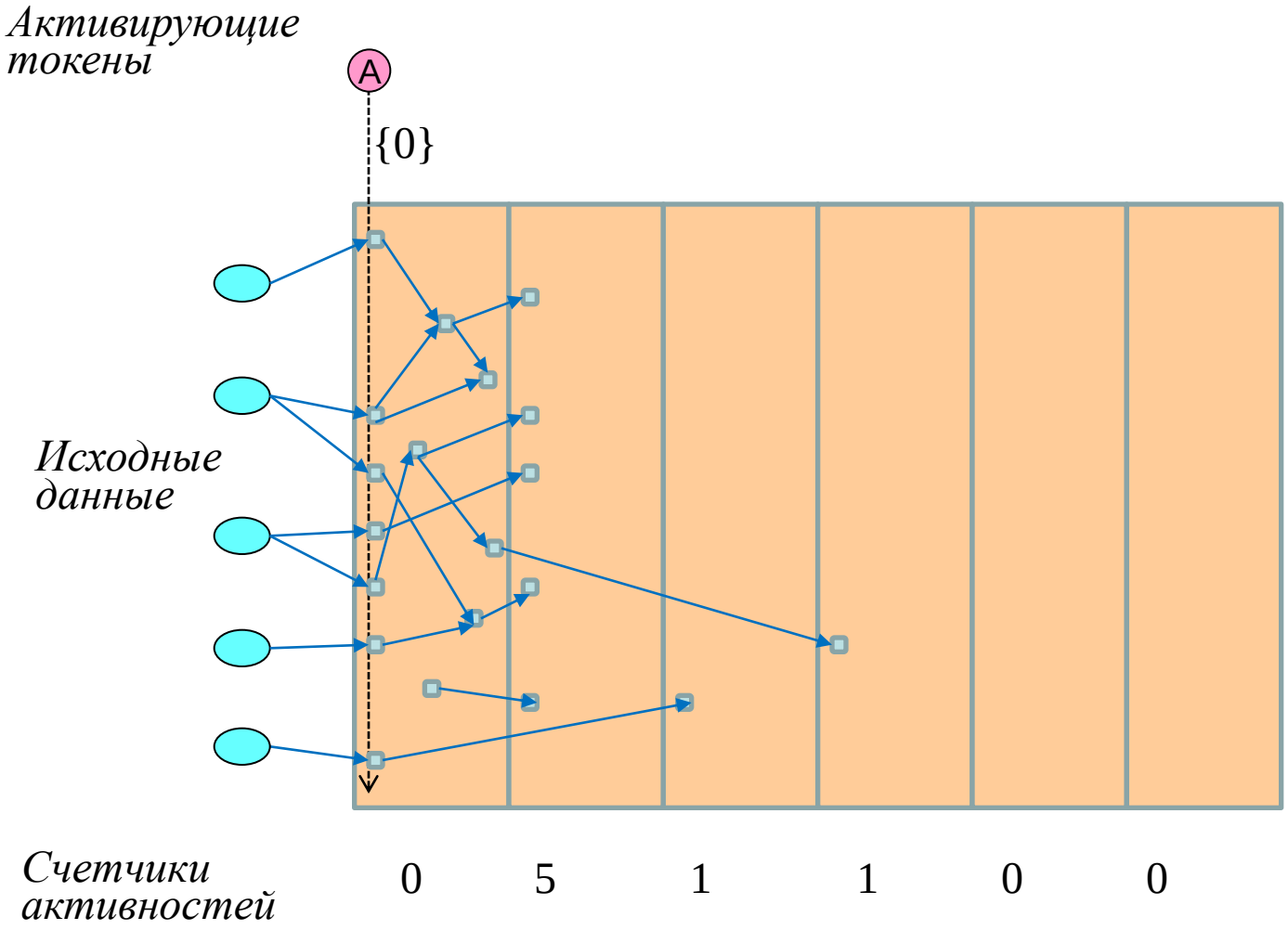


Исходные данные

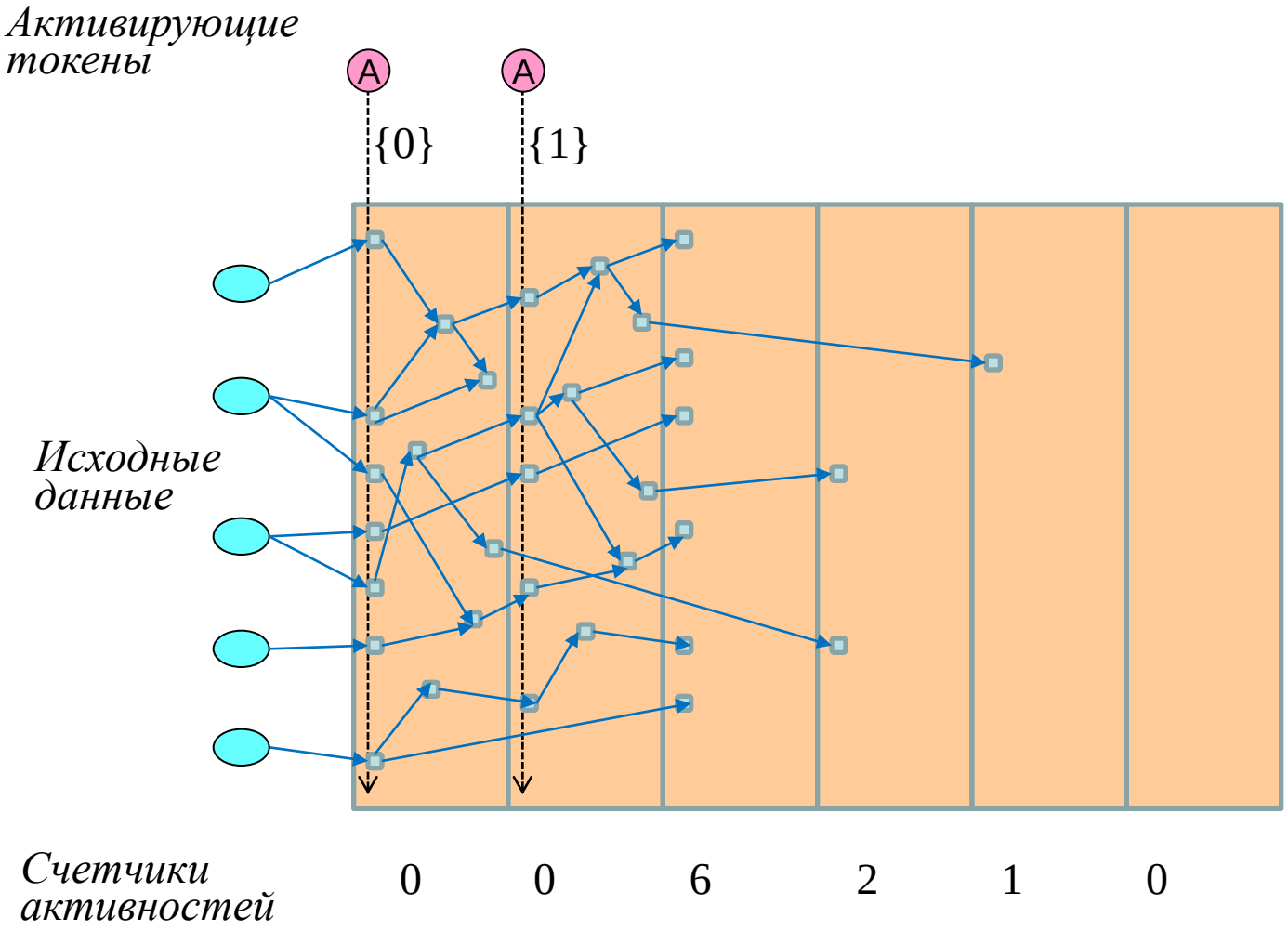
Счетчики активностей

7 0 0 0 0 0

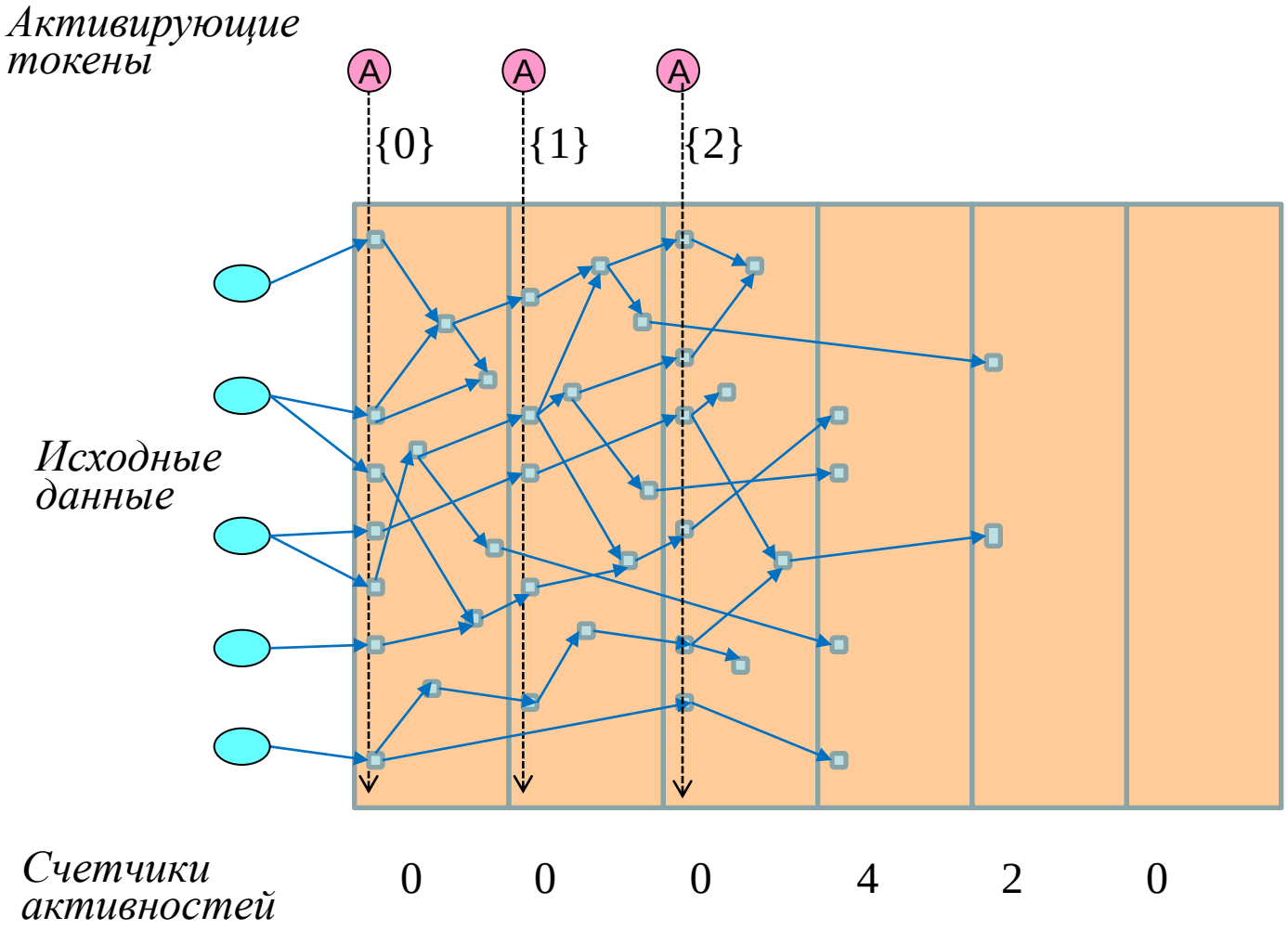
Ход вычисления под управлением МРТ



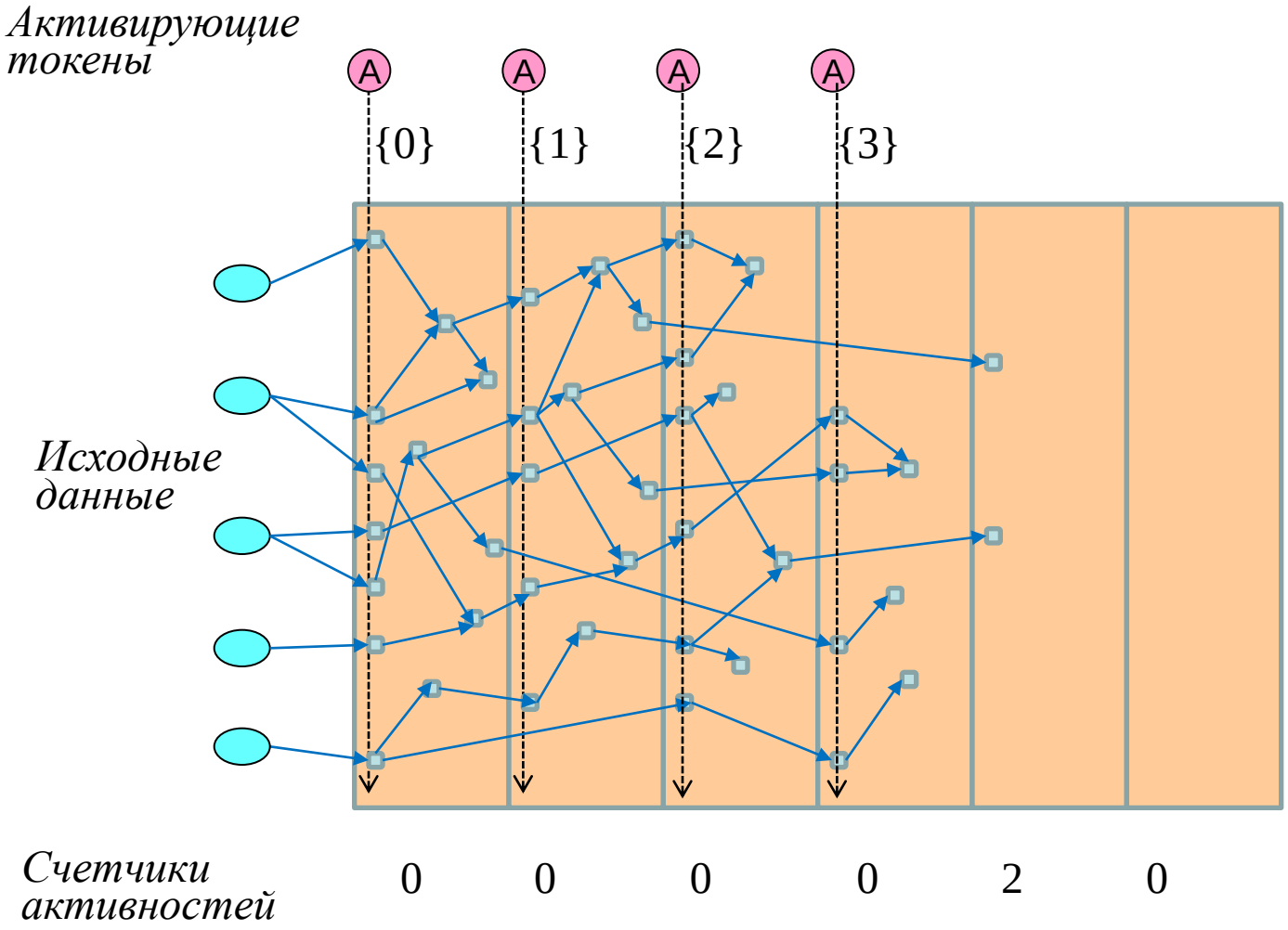
Ход вычисления под управлением МРТ



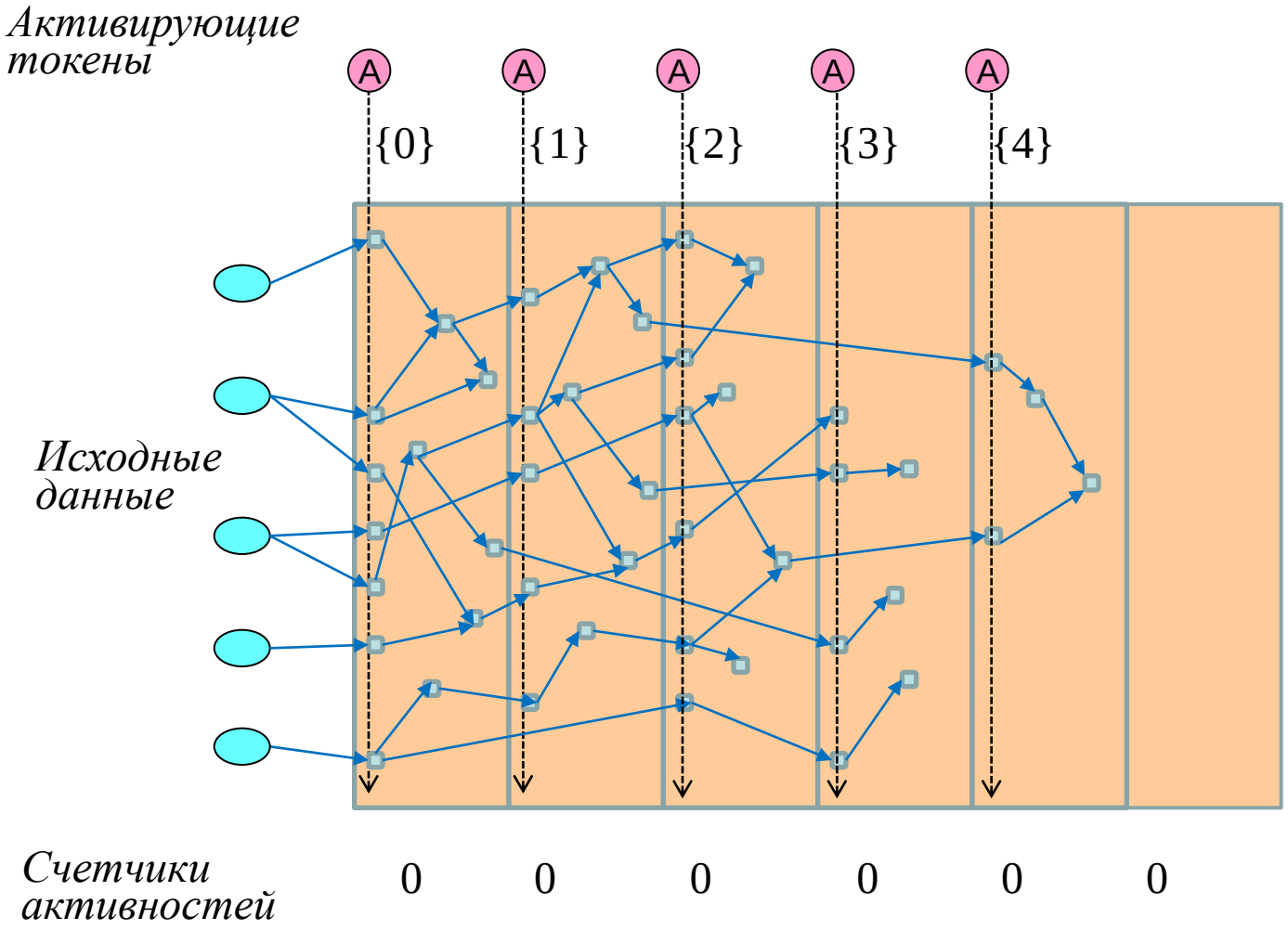
Ход вычисления под управлением МРТ



Ход вычисления под управлением MPT

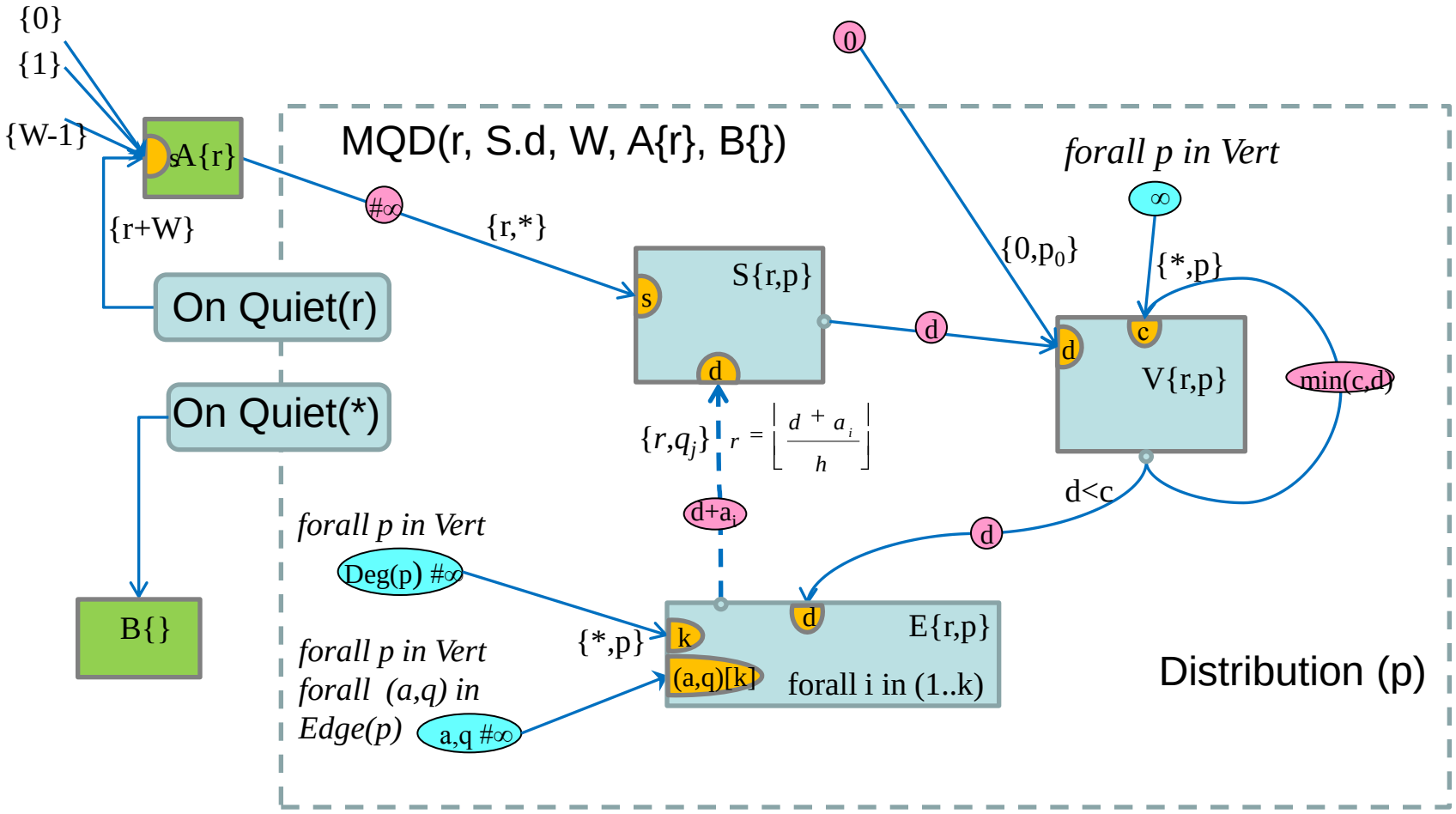


Ход вычисления под управлением МРТ



КОНЕЦ!

Схема алгоритма. Вариант с MPT.



Разрешающий токен S.s{*r*,*} можно посылать по завершению всякой активности в подпространстве {*r*-1,*}. Или {*r*-*W*,*}, где *W* =1,2,… – ширина окна активности.
Для этого исполняющая система должна иметь *многоуровневый детектор тишины.*
NB: здесь для токенов на узле S не требуется поиск по значению p.

Программа на PolyDFL

Исходные данные и инициализация:

k → E.k{*,*p*}	– для всех вершин p, из которых выходит k дуг
(*a*,*q*) → E.e{*,*p*};	– для всех дуг веса *a* из вершины *p* в вершину *q*
MAXREAL → V.c{*,*p*};	– для всех вершин *p* (признак недостигнутости)
0 → V.d{p_0};	– для начальной вершины p_0

Описания узлов на PolyDFL:

```
varconst  h:real; W:int;
MQD-area R{r} active(S.d) window(W);
        node V(d,c) {p};
          min(d,c) → V.c{*,p};
          if d<c then
            d → E.d{r,p};
        node E(sav e(a,q)[k], sav k, d) {p};
          for i :=1 to k do
            let v := a[k]+d in
              v → S.d{trunc(v/h),q[i]};
        node S(s,d/min) {p};
          d → V.d;
end
   activate(r) as MAXREAL → S.s{r,*};
   exit as 0 → B{};  -- вызывается по полной тишине
```

Выводы

- Потоковая модель вычислений в парадигме раздачи удобна для написания асинхронных программ обработки графов.
- Присущие ей встроенных механизмы передачи и хранения токенов адекватно поддерживают структуры данных (динамические множества, таблицы и т.п.), необходимые для работы таких алгоритмов.
- Принципы организации вычислений - активация по готовности - обеспечивают необходимые и удобные средства для локальной синхронизации независимых активностей.
- Для глобальной синхронизации имеется мощный аппарат распознавания тишины, избавляющий программиста от проблем. Аппарат распознавания многоуровневой тишины с окном ширины W удобен для организации глобального упорядочения. Варьируя размер шага h и ширину окна W, можно находить оптимальное решение, балансируя объем лишней работы и параллелизм.
- Пожалуй, для полного счастья не хватает средств локального упорядочения: выбора хранимого токена с минимальным значением.
- Нами реализован эмулятор, для которого программа кодируется в C. Пока в нем имеется только простой распознаватель тишины. Используя его в цикле, мы имитируем многоуровневый с окном W=1.

СПАСИБО